DM4 : Appartenance à un langage algébrique

Ce devoir est facultatif. Il est à rendre pour le 29/11. Les deux parties sont largement indépendantes et il est possible de ne rendre que la première. L'objectif du sujet est d'étudier deux algorithmes permettant de résoudre le problème de décision suivant (dit, problème du mot) :

> **Entrée :** Un langage algébrique L et un mot m sur un même alphabet Σ.
> **Sortie :** Oui si $m \in L$, non sinon.

Partie 1 *Algorithme de Cocke-Younger-Kasami*

Les fonctions à implémenter dans cette partie le seront en utilisant la syntaxe C. On supposera que les bibliothèques string.h, stdlib.h et stdbool.h ont été chargées.

Dans cette partie, on considère un langage algébrique L ne contenant pas ε décrit par une grammaire algébrique $G = (\Sigma, V, S, \mathcal{R})$ qu'on suppose mise sous forme normale de Chomsky. On rappelle que dans ce cas, les règles de G sont nécessairement de la forme $X \to a$ ou $X \to YZ$ avec $X, Y, Z \in V$ et $a \in \Sigma$.

Soit un mot $m = m_1...m_n \in \Sigma^+$. On cherche à savoir s'il appartient à $L = L(G)$. Pour ce faire, on introduit pour tous $i, j \in [\![1, n]\!]$ tel que $i \leq j$ l'ensemble $E_{i,j}$ des non terminaux de G qui engendrent le mot $m_i...m_j$. Notre objectif est donc de calculer $E_{1,n}$: l'axiome S en fait partie si et seulement si $m \in L(G)$.

1. Si $i \in [\![1, n]\!]$, expliquer comment déterminer l'ensemble $E_{i,i}$ à partir de G.
2. Si $i, j \in [\![1, n]\!]$ et $i < j$, montrer que :

$$E_{i,j} = \bigcup_{k=i}^{j-1} \{X \in V \mid X \to YZ,\ Y \in E_{i,k} \text{ et } Z \in E_{k+1,j}\}$$

On considère alors l'algorithme dynamique suivant :

```
Entrée : Une grammaire G sous forme normale de Chomsky tel que ε ∉ L(G) et un mot m ≠ ε.
Sortie : Vrai si m ∈ L(G) et faux sinon.
CYK(G, m) =
    Initialiser une matrice E = (e_{i,j})_{i,j∈[1,n]} remplie de ∅
    Pour i allant de 1 à n
        Pour toute règle de la forme X → a de G
            Si a = m_i alors e_{i,i} ← e_{i,i} ∪ {X}
    Pour d allant de 1 à n − 1
        Pour e_{i,j} sur la d-ème surdiagonale de E
            Pour k allant de i à j − 1
                Pour toute règle de la forme X → YZ de G
                    Si Y ∈ e_{i,k} et Z ∈ e_{k+1,j} alors e_{i,j} ← e_{i,j} ∪ {X}
    Si S ∈ e_{1,n} renvoyer vrai, sinon renvoyer faux
```

3. On considère la grammaire G_{ex} dont les règles sont données par :

$$S \to XY, \qquad T \to ZT \mid a, \qquad X \to TY, \qquad Y \to YT \mid b, \qquad Z \to TZ \mid b$$

   En utilisant l'algorithme précédent, déterminer si *abab* appartient à $L(G_{ex})$. On dessinera explicitement la matrice E et le contenu de ses cases en fin d'algorithme.

4. Justifier la correction de cet algorithme.

5. Déterminer sa complexité en fonction de $|m|$ et d'une autre grandeur pertinente.

La fin de cette partie fait implémenter l'algorithme de Cocke-Younger-Kasami, dont le pseudo-code est donné ci-dessus. Les terminaux seront un sous ensemble des caractères ASCII. Les non terminaux (aussi appelés variables) seront représentés par des entiers numérotés consécutivement à partir de 0. L'axiome portera systématiquement le numéro 0. Une règle dans une grammaire sous forme normale de Chomsky sera représentée à l'aide de la structure suivante :

```
struct regle{
    int type;
    int membre_gauche;
    char lettre;
    int variable1;
    int variable2;
};
typedef struct regle regle;
```

Le type d'une règle est un entier valant 1 si la règle est de la forme $X \to a$ et 2 si elle est de la forme $X \to YZ$. Dans les deux cas, le champ membre_gauche contient le numéro du non terminal X. Dans le premier cas, le champ lettre contient a et les champs variable1 et variable2, -1. Dans le deuxième cas, les champs variable1 et variable2 contiennent les numéros de Y et Z respectivement et le champ lettre un caractère ne faisant pas partie de l'ensemble des terminaux de la grammaire considérée.

Une grammaire sous forme normale de Chomsky sera représentée à l'aide de la structure :

```
struct grammaire{
    int nb_variables;
    int nb_regles;
    regle* productions;
};
typedef struct grammaire grammaire;
```

Si g est un objet de type grammaire représentant $G = (\Sigma, V, S, \mathcal{R})$, g.nb_variables correspond à $|V|$, g.nb_regles à $|\mathcal{R}|$ et g.productions est un tableau de taille g.nb_regles contenant les règles de G.

6. Indiquer les valeurs des champs nb_variables et nb_regles d'un objet de type grammaire représentant la grammaire G_{ex}. Après avoir précisé une numérotation des non terminaux, déclarer deux objets de type regle représentant les règles $S \to XY$ et $Y \to b$ de G_{ex}.

Pour représenter un ensemble de non terminaux d'une grammaire donnée sur un ensemble de variables V, on utilisera un tableau de taille $|V|$ rempli de booléens. A la case i, ce tableau contiendra true si la variable numéro i est présente dans l'ensemble et false sinon.

7. Ecrire une fonction bool CYK(grammaire* g, char m[]) implémentant l'algorithme de Cocke-Younger-Kasami. On expliquera les idées ne découlant pas directement de la lecture du pseudo-code.

8. Vérifier votre réponse à la question 4 à l'aide de cet algorithme puis déterminer si $m_1 = bbaabaabbbab$ et $m_2 = abaabaabbbab$ font partie de $L(G_{ex})$.

9. (bonus) Expliquer comment modifier l'algorithme CYK de façon à ce qu'il réponde correctement au problème du mot lorsqu'on lève les contraintes $\varepsilon \notin L(G)$ et $m \neq \varepsilon$ (on suppose toujours que G est sous forme normale de Chomsky).

10. (bonus) Implémenter la mise sous forme normale de Chomsky d'une grammaire.

Partie 2 *Analyse descendante*

Les fonctions à implémenter dans cette partie le seront en utilisant la syntaxe Ocaml.

On cherche toujours à résoudre le problème du mot. Pour une grammaire donnée, on note N l'ensemble de ses non terminaux (notés avec des majuscules) et T l'ensemble de ses terminaux (notés avec des minuscules). Les lettres grecques désignent des mots de $(N \cup T)^\star$ et le mot vide est noté ε.

A titre d'exemple, on se dote d'une version simplifiée G de la grammaire du langage de programmation LISP. L'ensemble de ses terminaux est $T = \{\mathsf{sym}, (,), \#\}$, l'ensemble de ses non terminaux est $N = \{S, L, E\}$, son symbole initial est S et ses règles sont :

$$\begin{aligned} S &\to L\# \\ L &\to \varepsilon \mid EL \\ E &\to \mathsf{sym} \mid (L) \end{aligned}$$

1. Pour un entier $n \in \mathbb{N}$ arbitraire, donner un mot de longueur au moins n engendré par la grammaire G et la dérivation à gauche correspondante.

Pour réaliser l'analyse syntaxique de ce langage, on se donne un type token pour les symboles terminaux. Ainsi, Sym représente le terminal sym, Lpar le terminal (, Rpar le terminal) et Eof le terminal #.

```
type token = Sym | Lpar | Rpar | Eof
```

Notre objectif est d'écrire une fonction accepts: token liste -> bool qui détermine si un mot, donné comme une liste de terminaux, appartient ou non au langage de la grammaire G. Pour cela, on commence par construire trois fonctions mutuellement récursives, une pour chaque non terminal de la grammaire, ayant les types suivants :

parseS: token list -> token list
parseL: token list -> token list
parseE: token list -> token list

La fonction parseX reconnaît un préfixe maximal de la liste passée en argument comme étant un mot dérivé du non terminal X et renvoie le reste de la liste. Si un tel préfixe n'existe pas, la fonction lève l'exception SyntaxError définie comme suit :

```
exception SyntaxError
```

Pour définir ces trois fonctions, on se donne un tableau à deux entrées appelé *table LL* et dont la construction sera explicitée par la suite. Cette table indique, pour un non terminal que l'on cherche à reconnaître et pour un terminal au début de la liste, la règle de production à utiliser. Voici la table pour notre grammaire :

	sym	(	)	#
S	$L\#$	$L\#$		$L\#$
L	EL	EL	ε	ε
E	sym	(L)		

La fonction parseS est définie à l'aide de la première ligne de cette table et voici son code :

```
let rec parseS l = match l with
  |(Sym|Lpar|Eof)::_ -> (match parseL l with
                          |Eof::q -> q
                          |_ -> raise SyntaxError)
  |[] |Rpar::_ -> raise SyntaxError
```

En particulier, une case vide dans la table LL est interprétée comme un échec de l'analyse.

2. Donner le code des fonctions parseL et parseE.

3. Donner le code de la fonction accepts.

On cherche à présent à construire une table LL pour une grammaire arbitraire. On introduit pour cela une première notation : on dit qu'un non terminal X est *nul*, et on note $\textsc{Nul}(X)$, si le mot vide peut être dérivé de X c'est-à-dire si $X \Longrightarrow^\star \varepsilon$

4. Indiquer quels sont les symboles nuls de la grammaire G.

Pour déterminer $\textsc{Nul}(X)$, on propose l'algorithme suivant :

1) Initialement, on fixe $\textsc{Nul}(X)$ à false pour tout non terminal X.
2) Pour chaque non terminal X, on affecte la valeur true à $\textsc{Nul}(X)$ s'il existe une production $X \to \varepsilon$ ou une production $X \to X_1...X_p$ avec les X_i des non terminaux tels que pour tout i on ait $\textsc{Nul}(X_i)$.
2) Si l'étape 2) a modifié au moins une valeur $\textsc{Nul}(X)$, on recommence l'étape 2).

5. Montrer que cet algorithme termine.

6. Montrer que cet algorithme détermine bien les valeurs de $\textsc{Nul}(X)$.

Pour un non terminal X, on définit à présent deux ensembles de terminaux :

- $\textsc{Premiers}(X)$ est l'ensemble des terminaux qui peuvent apparaître au début des mots dérivés depuis X : $\textsc{premiers}(X) = \{t \in T \,|\, \exists \alpha \in (N \cup T)^\star,\, X \Longrightarrow^\star t\alpha\}$.
- $\textsc{Suivants}(X)$ est l'ensemble des terminaux qui peuvent apparaître après X dans une dérivation : $\textsc{Suivants}(X) = \{t \in T \,|\, \exists \alpha, \beta \in (N \cup T)^\star,\, S \Longrightarrow^\star \alpha X t \beta\}$

7. Donner les ensembles $\textsc{Premiers}(X)$ et $\textsc{Suivants}(X)$ pour la grammaire G prise en exemple.

On admet que l'on peut calculer les ensembles $\textsc{Premiers}(X)$ et $\textsc{Suivants}(X)$ pour toute grammaire. On étend Nul et Premiers aux mots de $(N \cup T)^\star$ de la manière suivante ($t \in T$ et $X \in N$) :

$$\begin{array}{l}
\textsc{Nul}(\varepsilon) = \mathsf{true} \\
\textsc{Nul}(X_1...X_p) = \mathsf{true} \text{ si } \textsc{Nul}(X_i) = \mathsf{true} \text{ pour tout } i \\
\\
\textsc{Premiers}(\varepsilon) = \emptyset \\
\textsc{Premiers}(t\alpha) = \{t\} \\
\textsc{Premiers}(X\alpha) = \textsc{Premiers}(X) \text{ si } \textsc{Nul}(X) = \mathsf{false} \\
\textsc{Premiers}(X\alpha) = \textsc{Premiers}(X) \cup \textsc{Premiers}(\alpha) \text{ si } \textsc{Nul}(X) = \mathsf{true}
\end{array}$$

On construit la table LL de la manière suivante : pour un non terminal $X \in N$ et un terminal $t \in T$, on écrit la règle de production $X \to \gamma$ dans la case (X, t) de la table

- si $t \in \textsc{Premiers}(\gamma)$,
- ou si $\textsc{Nul}(\gamma)$ et $t \in \textsc{Suivants}(X)$.

On peut alors se servir de cette table pour réaliser une analyse syntaxique dès lors que chacune de ses cases comporte au plus une règle de production. On pourra vérifier que la table annoncée pour la grammaire G est correcte. On considère à présent une grammaire G' sur les mêmes symboles terminaux que G, un ensemble de non terminaux $\{S', L', E'\}$, un symbole initial S' et l'ensemble de règles suivant :

$$\begin{array}{l}
S' \to L'\# \\
L' \to \varepsilon \,|\, L'E' \\
E' \to \mathsf{sym} \,|\, (L')
\end{array}$$

8. Montrer que G' reconnaît le même langage que G.

9. Construire la table LL pour cette seconde grammaire. Permet-elle de coder un algorithme pour l'analyse syntaxique des mots générés par cette grammaire ?